Visibility as Central Abstraction in Event-based Systems

Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann

Department of Computer Science
Darmstadt University of Technology, D-64283 Darmstadt
{fiege,gmuehl}@gkec.tu-darmstadt.de
{mezini,buchmann}@informatik.tu-darmstadt.de

Abstract. We introduce scopes as basic abstraction in event-based systems. While existing work disregarded the role of an administrator and simply focused on using `pub` and `sub` primitives in flat design spaces, we devise on top of the visibility concept abstractions that support bundling and composing of new components, refining delivery semantics in these bundles, and mappings in heterogeneous systems.

1 Engineering Event-based Systems ...

Publish/subscribe or event notification services are used increasingly often in distributed systems. They offer the ability to easily compose varying sets of components, facilitating loose coupling and asynchronous operations. The existing notification services typically have simple APIs with plain semantics: `pub()` and `sub()` methods for publishing notifications and registering callbacks with subscriptions [2] that are called when matching notifications are published. The notification services implement simple distributed broadcast and filter mechanism, and they are used for asynchronous notification and exception handling purposes, often besides the 'normal' operation of request/reply-based distributed systems.

However, event-based cooperation can also be used for designing general purpose, highly configurable distributed systems, although the inherent complexity increases drastically if no further abstractions are available. The engineering support is not adequate and nowhere near the support we know for systems based on remote method invocation (RMI). For coordinating and composing loosely coupled systems there exists lot of related work in the area of Linda-like coordination systems [6] and composition languages, e.g., [10], but the characteristics of event-based systems are not fully supported.

We analyze the engineering requirements of event-based systems and propose a module construct for abstraction and encapsulation by applying the notion of scoping to event-based systems in [3]. The visibility of events and components is used as fundamental basis for designing and engineering these systems. From an engineering point of view, scopes offer a module construct for event-based systems, being an abstraction and encapsulation unit at the same time. As an abstraction unit, a scope provides the rest of the world with common higher-level input and output interfaces to the bundled subcomponents. As an encapsulation unit, a scope constrains the visibility of the notifications published by the grouped components. It hides the details of the composition implementation, such as the underlying data transmission mechanisms, the interface mappings that map between internal and external representations of notifications, security policies, transmission policies controlling the way notifications are forwarded, etc. The structure built thereby is orthogonal to the components' implementation, separating concerns of implementation and interaction. As defined in our model, scopes have the flavor of component frameworks in the sense of Szyperski [13]: they encode the interactions between components and can themselves act as components in higher-level frameworks.

In this position paper, the scoping concept is revaluated from the viewpoint of providing abstractions of communication between event-based components, reflecting the ongoing work on transforming a straightforward Java implementation of the basic features to a coordination and configuration language for event-based systems.

2 ... and the Role of Visibility

Software engineering research early identified information hiding and abstraction [12] as basic principles that have influenced the development of structured programming, modules, classes, and compo-

nents, all of which provide mechanisms to structure software systems. While being an integral part of request/reply-based distributed systems, e.g., CORBA [11], comparable hierarchical structuring mechanisms are missing in event-based systems. As a result, event-based systems are generally characterized by a 'flat design space': Subscriptions select out of *all* published notifications without discriminating producers. Any further distinctions are necessarily hard-coded into the communicating components, mixing application structure and component implementation. Loose coupling, the very feature of event-based systems, is thereby sacrificed.

2.1 What abstractions do we need?

Design, engineering, and administration of open distributed systems incorporate a multitude of different roles that are responsible for different aspects of the system. In event-based systems, we can identify an especially important role besides producers and consumers which do not know of each other: it is the administrator's task to combine and orchestrate the otherwise 'blind' event-based components so that the bundle accomplish a common functionality. However, typical implementation techniques of publish/subscribe systems concentrate on efficiency issues and overlook the need for effective support of appropriate programming abstractions. Beside the known `pub` and `sub` we additionally need abstractions supporting the role of the administrator. We identify the following requirements lying in this role's responsibility:

- Bundling of components
  It should be possible to bundle individual components into higher-level syntactical and semantical units, offering higher levels of abstraction and reusability. Locality, encapsulation, and composing existing units are well-known concepts for mastering complexity and support evolution.
- Heterogeneity
  A single uniform event notification service with uniform syntax and semantics will hardly be able to cope with the requirements of all parts of large distributed systems operating in heterogenous environments. We draw the requirement that bundling of related components should not only encapsulate functionality but also delimit common syntax and semantics.
- Flexible configurations
  Similar to the diverse requirements regarding data representation in heterogeneous environments, a static definition of notification transmission semantics is not adequate either.
- Support of activities
  The engineering of complex systems not only benefits from bundling related components according to application structure but also from identifying sessions of interdependent activities. This is especially important in event-based systems, where the identity of peers is unknown and communication is a priori stateless in the sense that consecutive notifications cannot be interrelated.

The requirements, although similar to those of request/reply-based systems, emphasize the role of the administrator since the mentioned aspects lie in the glue between a system's components.

2.2 Visibility and Scopes

Encapsulation is a prerequisite to system evolution [12] and the notion of visibility is widely used in software engineering as structuring technique in order to limit the impact of changing parts of the system.

In order to address the requirements stated in the previous section, we introduce the concept of *scopes* for decomposing event-based systems. A scope is *an abstraction that bundles a set of producers and consumers* in that the visibility of notifications published by a producer is confined to the consumers belonging to the same scope as the publisher. It can recursively be a member of other scopes. It offers a powerful structuring mechanism to group constituent components which belong together according to some criteria derived from the application structure and/or semantics. Vice versa, it defines locality that can be used to customize semantics in a discriminated part of the system and that provides an encapsulated module whose interaction with the remaining system can be explicitly controlled, localizing the relationships between components, outside of the components themselves. Scopes offer

the missing notion of a module in event-based systems, for bundling several components into a higher-level component.

Scoped event-based systems are modeled by a directed, acyclic graph $G = (C, E)$ (see Fig. 1) that describes the superscope/subscope relationship. The set of nodes C is comprised of simple components $\mathcal{C}$ and complex components $\mathcal{S}$, i.e., scopes. The edges E are a binary relation over C. An edge from node c_1 to c_2 in G stands for c_2 being a superscope of c_1. A more formal treatise is published in [5].

2.3 Controlling Visibility

Using the graph of scopes G given above, we define the *visibility* of components as a reflexive, symmetric relation v over C. Informally, component X is visible to Y iff X and Y have a common superscope. For a component X, let $\mathit{super}(X) = \{X' \mid (X, X') \in E\}$ denote the set of scopes that are direct superscopes of X. With the transitive closure super^* visibility is defined by $v(X, Y) \Leftrightarrow \mathit{super}^*(X) \cap \mathit{super}^*(Y) \neq \emptyset$. In the graph in Fig. 1, for example, $v(Y, U)$ holds but not $v(X, U)$.

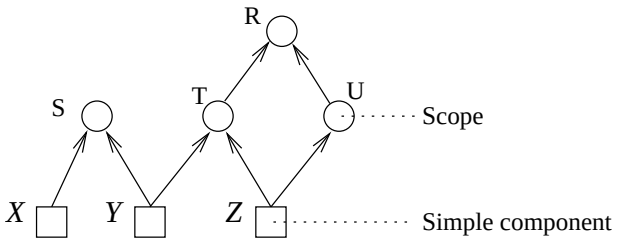

Fig. 1. A graph of components/scopes

As a primary structuring mechanism we enhance scopes with interfaces in order to compose its constituents into new components with its own interface. *Input* and *output interfaces* for components are defined by filters that determine the set of notifications allowed to cross a component's boundaries. Input filters are specified in subscriptions and output filters are issued in *advertisements* that define the set of notifications a component is able to publish. A filter $F \in \mathcal{F} := \{f \mid f(e) = e \ \vee \ f(e) = \varepsilon\}$ is a mapping function over the set of all possible notifications $\mathcal{N}$ plus the empty notification ε. Often, filters are defined as boolean functions returning `true` if a notification matches. In our model, we use a generalized form of filters that are allowed to pass matched events in an unchanged form. A notification n is either mapped to itself or to ε, indicating that n is matched or blocked, respectively. Allowing filters to pass matched events in an unchanged form facilitates filter composition: $(F_1 \circ F_2)(e) = F_1(F_2(e))$.

The semantics of event publication and delivery can now be refined: A notification is delivered to a consumer if (a) the producer and the consumer are visible to each other, (b) the notification matches one of the subscriptions previously issued by this consumer, and (c) the notification is allowed to pass all interfaces along the path of visibility in the graph.

3 The Role of Scopes

The scopes represent a unit of encapsulation that allows to bind further processing control and semantics to a delimited part of an application. We already have bound interfaces in a straightforward way to scopes and will sketch further extensions in the following.

Delivery and dissemination semantics can be refined on a per scope basis by introducing *delivery policies* that affect deliverable notifications produced in a superscope or by some constituent subcomponent and determines which members of the scope are to receive the notification. An example is a 1-of-n policy which delivers only to one out of a group of possible receivers. The idea of meta object protocols [8] of object-oriented programming languages is applied here in order to offer the ability to order, queue, redirect incoming messages. Policies can also be viewed in the opposite direction. A *publishing policy* controls publication into the direct superscopes. This selection is part of the scope and not

interwoven with the application functionality in simple components. Publishing policies are different from interfaces in that they operate on a per notification basis and might be used to delay notifications for a certain amount of time or until a condition becomes valid, for example.

A generalized form of filters, i.e., input and output interfaces, is allowed to transform events passing a scope boundary: The set of event mappings include the set of filters, $\mathcal{M} \supset \mathcal{F}$; for a further discussion we refer to [3, 5].

The described features of scopes are used to implement session scopes that facilitate having *and* differentiating multiple sessions, i.e. superscopes, simultaneously; a further discussion is excluded here due to space limitations.

A scope bundles a set of components and reifies a common context. The components need not to be explicitly aware of this context, but it distinguishes all notifications published within the scope from those published elsewhere. We use this idea by binding a set of name/value pairs with each scope, automatically add these pairs to all events published by the scope, and strip them from incoming events that originated in a superscope. As an additional filter, the context can also be used to discard all incoming events which, compared to the scopes context, carry a name/value pair with a matching name and mismatching value.

4 Enhancing Pub/Sub

From an engineering point of view, scope implementation is software composition: it orchestrates the composed entities. On the other hand, a scope addresses the implementation issues in distributed environments, e.g., localizing the implementation used to address the components, like broadcast, multicast, point-to-point connections, etc.

We have to distinguish the role of the producer/consumer and that of an administrator. Accordingly, the abstractions accessible for application functionality are those already known for unstructured event-based systems (e.g., [2]). They must be realized in the programming language used for implementing the application code. Currently, a component's implementation in Rebeca, our prototype of an event notification service [4], is required to inherit from the base class `Component` that offers `pub` and `sub` methods to access the notification service. In a sense this approach is a 'pure' solution since it requires all participating components to inherit from a specific superclass. However, other approaches using precompilers might offer a cleaner interface but generate a comparable implementation without enhancing expressiveness [2].

The abstraction describing the scopes themselves may use any other language which is suitable as configuration and composition language and allows to reconfigure running systems on-line. We distinguish three different, layered approaches: a visual configuration tool used to adapt the graph of scopes and their individual features, a composition language used to describe the scope implementation, and a Java implementation realizing the features. In general, we assume that a transformation from a visual to a compositional to an imperative language is possible, although simplicity and, in the second step, flexibility is degrading.

Basically, a scope describes an event stream shaping entity, controlling boundary passing data and directing the delivery within the boundary. Known solutions from both information flow (e.g., [7]) and composition languages (e.g., [1]) are eligible candidates for specification. System engineers benefit from the main feature of scopes of delimiting and localizing varying implementations in that not all 'types' of scopes need to be specified/implemented with the same language. The implementation of a given subgraph of scopes can be tailored to the specific needs of the respective scopes and their constituents. For this reason, the concept is evaluated with a naïve Java implementation, instantiating scopes as first-class objects and connecting components point-to-point. Every type of scope is implemented by a Java class, following the diagram in Fig. 2. The component interfaces are distinguish from the more general mappings although they are conceptually related because the former depends on the implementation of the underlying transport mechanism (JMS, TCP point to point, etc.) while the latter is scope dependent. Furthermore, the interfaces are part of the subscriptions and advertisements if the scopes are implemented on top of a generic pub/sub mechanism [9].

Any policy inherits from `DeliveryPolicy` the `destinations(event)` method that maps an deliverable event to a set of eligible destinations. The following sequence illustrates processing of an incoming event if we assume point-to-point connections to all components:

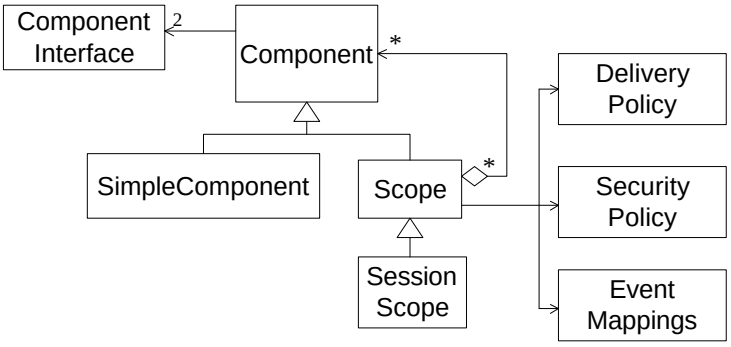

Fig. 2. The Meta-Model of the Scope Model

1. Apply input filters if the superscope uses unfiltered broadcast
2. Apply event mappings
3. Routing decision: generate eligible destinations based on the known input interfaces of constituent components.
4. Apply delivery policy to reduce the set of destinations
5. Transmit the data

As part of the scope implementation, methods for scope graph management, i.e., scope movement and deletion, are provided. We do not go into the details of security policies here but only introduce them as entities controlling access to the graph management.

Specific, predefined types of scopes may use exactly one static set of policies and even the scope interfaces may be determined at time of scope instantiation. A configurable scope type can be parameterized at run-time with changing policies; it simply implements a `ConfigurableScope` interface which lets an administrator externally provide new policy implementations.

5 Summary and Future Work

This position paper presented the notion of scopes as an abstraction of visibility in event-based systems and uses them as an extension of the well-known semantics of pub/sub systems. We specifically facilitate structuring and creation of event-based components and the management of relations between components outside of the components themselves. In addition to a native Java implementation, existing abstractions like composition languages and information flow specifications can be used and tailored to realize a module construct for event-based systems. Clearly, investigating and combining the different forms of scope-based programming in visual, compositional, and classical imperative languages is future work.

References

1. Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.
2. Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming Systems Languages and Applications*, pages 254–269, Tampa Bay, FL, USA, 2001. ACM Press.
3. Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering event-based systems with scopes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS, 2002. to be published.
4. Ludger Fiege and Gero Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.

5. Ludger Fiege, Gero Mühl, and Felix C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.
6. David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
7. J. Huang, A.P. Black, J. Walpole, and C. Pu. Infopipes – an abstraction for information flow. In *ECOOP 2001 Workshop on the Next 700 Distributed Object Systems*. Springer-Verlag, 2001. Also available as OGI technical report CSE-01-007.
8. Gregor Kiczales, Jim des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, MA, USA, 1991.
9. Gero Mühl, Ludger Fiege, and Alejandro Buchmann. Filter similarities in content-based publish/subscribe systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *Lecture Notes in Computer Science*, pages 224–238. Springer-Verlag, 2002.
10. Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, volume 924 of *LNCS*, pages 147–161. Springer-Verlag, 1995.
11. Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Version 2.3. Object Management Group, Framingham, MA, USA, 1998.
12. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
13. Clemens Szyperski. *Components Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.